INFO0004-2: A Mathematical Artist

Pr Laurent MATHY, Cyril SOLDANI

April 1, 2021

1 Introduction

In this project, you will implement a C++ *mathematical artist* (a.k.a. "martist").

Our martists will construct complex expressions from 6 basic expressions: x, y, $\sin(\pi \times expr)$, $\cos(\pi \times expr)$, the product of two expressions, and the mean (average) of two expressions. Martists thus represent complex expressions as nested basic expressions.

Note that for all values of x and y in $[-1, 1]$, all basic expressions have values in $[-1, 1]$. Therefore, any complex expression built from these basic expressions will also have values in $[-1, 1]$.

Using simple scaling, an *intensity level* in $[0, 255]$ can thus easily be obtained for any point (x, y) in the $[-1, 1] \times [-1, 1]$ square. Likewise, by scaling operations on x and y, this 2-by-2 square can be stretched into an w-by-h image, where w is the width and h is the height of the image (in pixels).

A colour image can be obtained by using 3 expressions, one for each colour component (red, green and blue) of the image.

2 Expressions

Of the six basic expressions, two are simple expressions (x and y), the four others are composed expressions (i.e., expressions made up of two or more expressions). All expressions, complex or basic, can thus be seen as trees whose internal nodes are composed expressions and whose leaves are simple expressions.

To control the complexity of the generated images, the *depth* of an expression is defined as the maximum permitted depth of the expression tree. For example, x has a depth of 1, $\sin(\pi y)$ has a depth of 2 and $\cos(\pi(x \times \sin(\pi y)))$ has a depth of 4.

You martist must be able to generate random expressions of a given depth. Note that the given depth only defines the *allowed* maximum complexity, i.e. any expression of a given depth is also a valid expression of a bigger depth. For example, y is an expression of depth 1, but is also a valid expression of depth 4 (and of any depth ≥ 1). The artistic *style* of your martist will depend on your interpretation of "generate a random expression of a given depth".

Your martist must also be able to read and write expressions using the format of table 1. This format uses *Reverse Polish Notation* (RPN), i.e. operands come before the operation. For example, expression $\cos(\pi(x \times \sin(\pi y)))$ would be printed as `xys*c` while input `xyays*` would give expression $\text{avg}(x, y) \times \sin(\pi y)$.

To read/write an image spec from/to an I/O stream, each colour component will be on a line, in the order red, green and blue. Each line must be terminated by a line feed character (ASCII character 10, denoted `'\n'` in C++).

Expression	Textual representation
x	`x`
y	`y`
$\sin(\pi e_1)$	e_1'`s`
$\cos(\pi e_1)$	e_1'`c`
$e_1 \times e_2$	$e_1' e_2'$`*`
$\text{avg}(e_1, e_2) = (e_1 + e_2)/2$	$e_1' e_2'$`a`

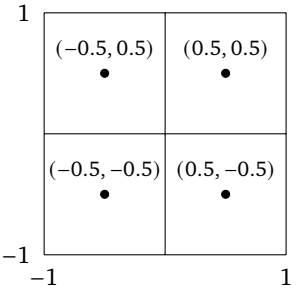
Table 1: Textual format for martist expressions, where e_1, e_2 are expressions, and e_1', e_2' are their corresponding textual representations.

3 Images

An image is represented by an array of pixels, that will be stored as a row-first one-dimensional array, with each pixel consisting of three 8-bit colour components (red, green and blue, in that order).

When you compute the mapping between image points and points in the 2-by-2 square, it is important to remember that these two *spaces* have their origins at different places: in $[-1, 1] \times [-1, 1]$, the origin $(0, 0)$ is at the center of the square; for the image, the origin $(0, 0)$ is the top-left corner. **The mapping between the square and the image is positional**: the top-left corner in the square must map onto the top-left corner in the image, the centre of the square must map to the centre of the image, etc.

As pixels make a discrete partition of the image space, you will use for each pixel the colour values for the center of that pixel. For example, a 2-by-2 image would use the following coordinates for (x, y) in the $[-1, 1] \times [-1, 1]$ square:



4 The `Martist` class

You will write the C++ `Martist` class that exposes the following interface:

- ```
  Martist(std::uint8_t *buffer, std::size_t width,
          std::size_t height, std::size_t redDepth,
          std::size_t greenDepth, std::size_t blueDepth)
  ```

  where `buffer` is a buffer of pixels as described above; `width` and `height` are the respective dimensions of the image (in pixels); `redDepth`, `greenDepth`, and `blueDepth` are the allowed depths for expressions representing the respective colour components. Note that if the depth of an expression is zero, the corresponding expression must have a constant value of 0.
  ```

- `void redDepth(std::size_t depth)`
  `std::size_t redDepth() const`

  allows to set and get the maximum allowed depth of the expression for the red component. You will also implement similar accessors for the green and blue components.

- `void changeBuffer(std::uint8_t* buffer, std::size_t width,`
  `std::size_t height)`

  changes the image buffer.

- `void paint()`
  generates a new (random) image and paint it to the buffer. All computations are done in double precision.

- `void seed(unsigned int seed)`
  changes the Martist's mood, or in other words, seeds the randomness. For a given *seed*, the sequence of images generated by `paint()` should always be the same.

- `std::istream& operator>>(std::istream &in, Martist& martist)`

  allows to read an image spec (see above) from a stream, updating the depths of expressions and redrawing the image buffer.

- `std::ostream& operator<<(std::ostream &out, const Martist& martist)`

  allows to write last painted image spec to a stream. Writing a martist that has not yet been painted is undefined.

You are free to design whatever private interface you need for your `Martist` class, and whatever other classes or functions that you see fit. However, your martist will only ever be accessed through the public interface described above, which must be defined in a `Martist.hpp` file. You can add other files as you see fit.

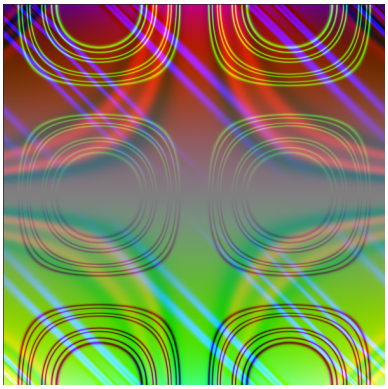
It is always the programmer's responsibility to make sure the provided image buffer is big enough. Your program can crash if passed a bogus buffer. It is also the programmer's responsibility to free that buffer.

The only two operations that should draw an image to the buffer are reading an image spec from an input stream and calling `paint()`. Other operations such as setting a depth should only only influence the next `paint()` operation.

You may want to write a simple test application to display your martist's images on screen. The CImg Library (`http://cimg.eu`) can help. Alternatively, you can also generate images in a simple format such as PPM. This is entirely optional, and won't be evaluated.

Here is an example of an image spec and corresponding image:

```
xy*cyxyya*sccsa*
yxsyc*ccccc*
yy*xyasscccs*
```



5 Remarks

5.1 Respect the interface

You **must respect the interface** given above.

Try to apply object-oriented concepts to your interface, so that your implementation could be easily changed or extended later on without breaking user code.

5.2 Readability

Your code must be readable:

- Make the organisation of your code as obvious as possible. Remember that you can create as many auxiliary functions, classes and files as you see fit.
- Complement your self-documenting code with comments, where appropriate.
- Choose a coding convention, and stick to it. *Consistency* is key.

5.3 Warnings

Your code must compile **without error or warning** with `g++ -std=c++17 -Wall` on the `cffs/oopp` Docker container, or on the `ms8??` machines. However, we advise you to check your code also with `clang++ -std=c++17 -Wall -Wextra` or tools like `cppcheck`.

5.4 Efficiency and robustness

Your code must be robust. The `const` keyword must be used correctly, sensitive variables must be correctly protected, memory must be managed appropriately, your code must not cause a crash (unless the user provides a bogus buffer).

Your code should be reasonably efficient and should not leak resources.

A tool such as valgrind can help to check your code behaves properly.

5.5 Code organisation

You are free to structure your program as you see fit, using multiple files if deemed useful. You will **at least** provide `Martist.{h,c}pp`, and a `Makefile` that allows to compile your files. That `Makefile` should support separate compilation (i.e. each C++ implementation file should be compiled separately to its own object file), and **must** build a `martist` application when we issue `make`, provided that the user adds its own `main.cpp` file containing the `main()` function.

You are expected to develop a `main.cpp` file to test your martist, but that file should not be submitted, and will not be evaluated. We will provide our own `main.cpp` file with our own `main()` function using your martist through the interface defined above.

5.6 Evaluation

Your code will be evaluated based on all the above criteria. Failure to comply with any of the points mentioned in **bold face** can (and most often will) result in a mark of zero!

6 Submission

Projects must be submitted through the submission platform before Monday **April the 26th**, 23:59 CET. After this time, a penalty will be applied to late submissions. This penalty is calculated as a deduction of $2^n - 1$ marks (where n is the number of started days after the deadline).

You will submit a `s<ID>.tar.xz` archive of a `s<ID>` folder, where `s<ID>` is your ULiège student ID. That folder should contain `Martist.{h,c}pp`, and any other C++ file required by your martist and your `Makefile`. It **should not** contain any `main()` function, we will provide our own in a `main.cpp` file when we test your submission.

The submission platform will do basic checks on your submission, and you can submit multiple times, so check your submission early!